3D PROGRAMMING, DV1542/DV1568: ASSIGNMENT 2

—

HELLO TRIANGLE

Joakim Ståhle-Nilsson
Blekinge Tekniska Högskola

2020ht, lp2

The purpose of this assignment is to learn the basics of rasterized rendering and the D3D11 API:

- *Introductionary usage of the D3D11 API*
- *Basic rasterized rendering using the graphics pipeline*
- *Managing and using simple resources used for rendering*

Preparation 1. Read the lab instructions

Read the *whole* document containing instructions for what needs to be done for the assignment.

End of preparation 1.

Preparation 2. Plagiarism and cooperation

The lab is to be done either individually or in groups of two students. While discussion and such is encouraged in the course, your implementation should be done by you/your group.

End of preparation 2.

Preparation 3. Preparative materials

Look through the material on the canvas page regarding rasterized rendering and use the sources that you find useful. Other sources are acceptable as well, just make sure that they are trustworthy.

End of preparation 3.

1 Assignment description

In this assignment you will create a simple D3D11 application that will render a quad using two triangles. The quad should be textured, be rotating around the Y-axis with an offset, and the Phong reflection model should be applied to it. There needs to be at least one light source in the scene that affects the quad visibly. You may use the provided demo code available on the Canvas page of the course as a base for the assignment, but you need to be able to explain it as well as all code you add. Figure 1 show an example of how the quad could look in the end. In it the quad has been textured with an image of a cat, and the phong reflection model has been applied (while not necessarily obvious at first glance without any movement there is a specular highlight at the cat's cheek).

1.1 General requirements

- Names should be descriptive, in English, and the style of the code should be consistent throughout the whole implementation.
- Variables should only be global if there is a very good reason for it.
- Remember to use const and references for parameters when appropriate.
- The code should be able to compile using either Visual Studio 2017/2019, or by using g++. If g++ is used then a makefile should be provided that can compile the program. Visual Studio is recommended but not enforced.
- The `C++11`/`C++14`/`C++17` standards should be used.
- No external libraries (unless mentioned in this document) should be used or needed in your implementation.
- DirectXMath is allowed for the mathematical aspects on the CPU side but you need to be able to explain the math itself.

Figure 1: Example of how the quad could look

1.2 Suggested steps

This section contains some general steps that can be used as a guideline for what needs to be done overall.

- Create a Win32 program that can open a window.
- Create the device, immediate device context, and swapchain.
- Create a render target view using the backbuffer.
  - At this point you should be able to test that you can clear the render target and present it, seeing a window filled with the clear colour.
- Create a texture to be used for the depth stencil as well as a depth stencil view based on the texture.
- Create a viewport that will cover the whole render target.
- Write basic vertex and pixel shaders that perform minimum operations, and load the .cso files into your program.
- Create an input layout based on your vertex shader.
- Create vertex data (position, UV, normal) for a quad and load it into a vertex buffer.
  - At this point you should be able to test that you can draw a simple triangle/quad that is already in clip space in the vertex buffer.
  - This is the point where the rasterizer demo approximately ends if you are using it as a base.
- Create a world matrix that perform the offsetting and rotation (tips: there exist functions in DirectX11 to help with this).
- Create a view matrix (tips: there exist functions in DirectX11 to help with this).
- Create a perspective projection matrix (tips: there exist functions in DirectX11 to help with this).
- Create constant buffer(s) to be able to bind the matrices to the vertex shader.
- Modify the vertex shader so that it uses the matrices bound through constant buffer(s) to transform vertices.
  - At this point you should be able to test that you can correctly transform the quad and that the vertices no longer need to be in clip space in the vertex buffer already.
- Load a texture from file and create a shader resource view for it so you can bind it to the pixel shader.
- Create a sampler state that can be bound to the pixel shader
- Modify the pixel shader so that it samples from the texture
  - At this point you should be able to test that you have a textured and transformable quad
- Create lights and a constant buffer(s) for them and other data that is necessary for the pixel shader to correctly be able to calculate lighting and perform shading operations.
- Modify the pixel shader so that it uses the phong reflection model.

2 Requirements

The following is a list of requirements that your program **must** meet before your turn it in for assessment.

- The scene contains two triangles that are joined into a quad that is rendered every frame.
  - The quad does not have to be a square but needs to reasonably fit within the default view area even when rotated.
  - UV coordinates should be so that the texture applied to the quad is visually correct.
  - The normal of the quad should be correctly perpendicular to the quad.
  - It is allowed (but not required) to use an index buffer.
- The quad is rotating around the Y-axis with an offset. That is, it is rotating around an arbitrary point like how planets rotate around a sun.
  - The rotation needs to be continuous and there should not be any limit to how many times it can rotate.
  - Rotation speed should be based on the time it takes to produce a frame so that it is independent of the frame rate.
  - It should be rotating fast enough that one can easily see the rotation, but not so fast that it makes it hard to see the quad or the lighting effect.
- A correct world, view, and projection transform needs to be applied to the quad each frame.
  - Perspective projection needs to be used.
  - The transform needs to be applied in the vertex shader, and the data provided by a constant buffer.
- At least one light source must affect the quad during at least some part of its rotation and used with the Phong reflection model.
  - The effect must be clearly visible during some part of the rotation and not hard to spot.
    - Consider how you position the light and camera so that you do not end up in a situation where you either do not see the effect, or for example the diffuse component never changes noticeably.
  - Lights may be either point or spot lights.
  - You may decide the colour of the light(s) on your own (consider the visibility of the lighting effect however).
  - It is allowed (but not required) to have more than one light in the scene and they must not all be of the same type or colour, but all calculations must be correct.
  - Lightning should be calculated each frame in the pixel shader
  - Necessary data (light data, camera pos, etc) needs to be provided to the pixel shader using a constant buffer.
- Your program may **not** have any unreleased COM objects when it finishes execution. To check this you can create the device with the `D3D11_CREATE_DEVICE_DEBUG` flag. You can then look at the output in visual studio and see if there were any unreleased COM objects after the program is finished.

You however do not have to consider the following.

- You do **not** have to consider or implement an intractable camera.
  - It is enough that the camera position and orientation can be changed by making changes to the code and recompiling it.
- You do **not** have to consider or implement any type of vertex loading from files. It is ok to hard code the vertices in the code.
- You do **not** have to consider or implement your own texture loading from files. You may use the one available in stb_image (that code must then be submitted on canvas as well).
- You do **not** have to use any specific image when texturing your quad. It should however be clear how it is oriented, and should not be inappropriate (use common sense). Make sure the image is submitted with the rest of the assignment.

3 Submission

Your submission should be a .zip file containing all the implementation files (header and source files), along with any other file that is necessary for compilation and/or execution (for example a makefile if g++ is needed).

If you have worked on the assignment in a group, then both students need to hand in the code on the Canvas page of the assignment. In addition, the name of who you have been working with needs to be clearly stated in a comment on the Canvas page.

4 Assessment

After you have submitted your code to the Canvas page you will have to first orally present your implementation during a lab session for either a teacher or a teaching assistant. This will be done individually (even if you have worked in a group) to make sure that everyone that is assessed has an understanding of what they have implemented and how it works. This presentation will be handled through either Zoom or the Discord server by using voice chat and screen sharing. The presentation should be around 10-15 minutes and thorough enough so that the person presented to can get a clear understanding of how the program works, without getting into every detail possible. No PowerPoint or similar tool is needed, as the code and discussion should be enough. The teacher/teaching assistant may also ask the student to go into more detail about, or ask questions regarding, certain parts of the implementation.

If the oral presentation is deemed to be insufficient then the student will have to present again at a later time. It is up to the teacher/teaching assistant to determine if the problem was small enough that a new presentation can be done at the next lab session, or if the student needs to take more time to be able to sufficiently present.

Once the oral presentation has been passed the teacher/teaching assistant will put a comment on the submission in Canvas to mark that the student has passed the oral presentation. After that a teacher will at some point perform a short review of the code. If nothing needs to be fixed after the review then the assignment will be graded as passed. There might however also be need for some fixing based on what was found in the review. If anything needs to be fixed then the assignment will be marked as Ux. The student then needs to perform changes based on the received feedback and then resubmit, at which point the code will be looked at again.

5 Hints

This section contains some helpful hints about the assignment that are worth considering and/or keeping in mind.

- The standard library chronos (`https://en.cppreference.com/w/cpp/chrono`) can be used to measure the frame/delta time.
- The stb_image library can be found here and a minimal example of reading a texture can be found here
- Use pen and paper if you hard code your vertices to make sure you get the order and positions correct
- A graphics debugger can allow you to both inspect data on the shader (for example contents of constant buffers and input/output from shader stages) and even debug the shader invocations of a frame
  - Visual Studio has a graphics debugger built in.
  - RenderDoc is an open source project for graphics debugger that has seen a lot of use.
  - There exist other options as well.
- Try and keep your code tidy and easy to work with, but do not put to much time into generalizing or abstraction.
  - The project will be more appropriate for more planned design.
- Try and work in steps so that you can confirm that things work during development and so that you do not have to try and fix every potential problem in one go at the end.
- If you create your device with the debug flag then you will get output in Visual Studios in most cases when something erroneous is detected. The output appears in the 'output' sub-window in Visual Studio.
- Win32 programs usually don't have a console for printing output. You can use `OutputDebugString()` instead. The output appears in the 'output' sub-window in Visual Studio. You can also read debug output outside of Visual Studio using the *DebugView* program from Microsoft.